Amendments to the Specification:

Please replace the paragraph on page 1, beginning on line 7, with the following paragraph.

This invention is directed to a method and apparatus for object-to-object [[Java]]JAVA Native interfacing. More particularly, the present invention is directed to a method and apparatus for facilitating the direct implementation of native or legacy code in the [[Java]]JAVA runtime environment.

Please replace the paragraph on page 1, beginning on line 22, with the following paragraph.

Although virtual machines are of a quite old computing concept, it wasn't until recently that commercial vendors were able to actually construct machines that are commercially viable and contain the runtime capabilities to scale as needed. Sun Microsystems and their [[Java]]JAVA platform are considered to be the first such vendor and environment, e.g., Sun [[Java]]JAVA 2 Native Interface.

Please replace the paragraph on page 2, beginning on line 1, with the following paragraph.

[[Java]]JAVA Native Interface or JNI, is designed with the objective of being generic enough that it enables [[Java]]JAVA applications to invoke any native code without regard to the native code's programming language. This rationale, for example, led to the tools provided for in JNI bridge development producing C language functions, which is by far the most generic and native-friendly programming language.

Please replace the paragraph on page 2, beginning on line 6, with the following paragraph.

In [[Java]]JAVA programming, an object is a software bundle of related variables and methods. Software objects all have state and behavior. Each software object maintains its state in

one or more variables. A variable is an item of data named by an identifier. A software object implements its behavior with methods, whereas a method is a function (subroutine) associated with an object. To clarify, everything that a software object knows (state) and can do (behavior) is expressed by the variables and methods within that object. Variables, formally known as instance variables, contain the state for a particular object. Instance methods inspect or change the state of a particular instance.

Please replace the paragraph on page 2, beginning on line 14, with the following paragraph.

An object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Encapsulation is used to define the packaging of an object's variables within the protective custody of its methods. In [[Java]]JAVA, an object can specify one of four different access levels for each of its variables and methods. These levels determine which other objects and classes may access that particular variable or method. A class is a blueprint, or prototype that defines the variables and methods common to all objects of a certain kind, thus, an object is typically an instance of some class. To state another way, a class consists of a group of related methods and variables lumped together under one name.

Please replace the paragraph on page 2, beginning on line 22, with the following paragraph.

In C++, objects are normally identified by their address (pointer). This address, if used improperly, can cause severe and hard-to-find damage to the program. Examples of such improper usage are double-free, stale pointers usage, and the like. The "smart pointer", or object reference management, replaces the object pointer with an object reference, a separate lightweight object which is managed independently from the object, much like object references in [[Java]]JAVA. Class Loaders use these pointers to locate the corresponding class file on disk, read the file into RAM and call `java.lang.ClassLoader.defineClass`, which tells the system to treat the RAM image as legitimate byte codes.

Please replace the paragraph beginning on page 2, line 30, and continuing to page 3, line 4, with the following paragraph.

Some object-oriented languages require that a programmer keep track of all the objects the programmer creates and to explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error prone. The [[Java]]JAVA platform uses garbage collection, which allows programmers to create as many objects as desired (limited, of course, by what the user's system can handle), and without worrying about destroying them. The [[Java]]JAVA runtime environment deletes objects when it determines that they are no longer being used.

Please replace the paragraph on page 3, beginning on line 5, with the following paragraph.

When there are no more references to a particular object, that object is eligible for garbage collection. References that are held in a variable are usually dropped when the variable goes out of scope. Or, a programmer or user may explicitly drop an object reference by setting the variable to the special value null. A program may have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection. The [[Java]]JAVA runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced.

Please replace the paragraph on page 3, beginning on line 12, with the following paragraph.

The majority of legacy code or code that [[Java]]JAVA must communicate with is actually written in C++ or some other object-oriented language. By its nature, C++ is very close concept-wise to [[Java]]JAVA. Both object-oriented languages use classes, objects, native class factories, and the like. In point of fact, C++ and [[Java]]JAVA are so close that they share common language constructs such as keywords, syntax, etc. However, JNI provides only C language binding and lacks the binding required for the C++ language.

Please replace the paragraph on page 3, beginning on line 18, with the following paragraph.

Thus there is a need for a method and apparatus to bridge the gap between the [[Java]]JAVA object world and the world of C++ objects.

Please replace the paragraph on page 3, beginning on line 21, with the following paragraph.

In accordance with the present invention, there is provided a method and apparatus for implementing native code directly in C++ with object structure very closely resembling object structure of [[Java]]JAVA. The method provides for mapping [[Java]]JAVA objects to C++ objects through the [[Java]]JAVA Native Interface without interfering with the object model, performance, or other characteristics of the implementation. The present invention also includes a configurable C++ garbage collector that compensates for [[Java]]JAVA garbage collection deficiencies.

Please replace the paragraph beginning on page 3, line 27, and ending on page 4, line 5, with the following paragraph.

Further in accordance with the present invention, there is provided a method for mapping a [[Java]]JAVA object to a C++ object, the steps comprising retrieving a value for a jobject. Next, a determination of the value of the jobject is made, where a nonzero value indicates a valid C++ object pointer and a zero value indicates that the jobject has not yet passed into the C++ boundary. The jobject instance is used to acquire the [[Java]]JAVA class name of the jobject. The class name is passed to a class loader and asked to return a corresponding class factory. Initializing the class factory and instantiating a native side object corresponding to the original jobject received. Finally, the C++ object type and method are resolved at compile time using a JNI wrapper routine.

Please replace the paragraph on page 4, beginning on line 6, with the following paragraph.

Still further in accordance with the present invention, there is provided a method for mapping a C++ object to a [[Java]]JAVA object, comprising the steps of returning a reference to a C++ object as a smart pointer. Next, the smart pointer is queried for an object ID, or key. The key is compared with an existing map to determine whether or not the key needs to be created. A value in the map that corresponds to the key is retrieve and is in the form of a jobject. The jobject ID is then returned.

Please replace the paragraph on page 4, beginning on line 17, with the following paragraph.

Further in accordance with the present invention, there is provided a method for collecting native side garbage, the steps comprising awakening a garbage collector, iterating an object map for weak references, collecting native side smart pointers, and decreasing the reference count of the native side object referred to by the smart pointer, wherein all native side objects that have their corresponding [[Java]]JAVA side counterparts out-of-scope are removed.

Please replace the paragraph on page 5, beginning on line 6, with the following paragraph.

Fig. 3 is a block diagram illustrating the illustrating the maintaining of an object reference of a C++ object within the instance data of a [[Java]]JAVA object;

Please replace the paragraph on page 5, beginning on line 8, with the following paragraph.

Fig. 4 is a block diagram illustrating the maintaining of a map from an object reference of a [[Java]]JAVA type object to a C++ type object;

Please replace the paragraph on page 5, beginning on line 10, with the following paragraph.

Fig. 5 is a flow chart depicting the mapping of [[Java]]JAVA to C++ and invocation of a method;

Please replace the paragraph on page 5, beginning on line 11, with the following paragraph.

Fig. 6 is a flow chart depicting the mapping of C++ to [[Java]]JAVA and the returning of an object ID;

Please replace the paragraph on page 6, beginning on line 2, with the following paragraph.

The present invention is directed to an apparatus and method for providing a natural map for movement of objects from one language to another. More particularly, the present invention is directed to a method and apparatus for mapping object-to-object using the [[Java]]JAVA Native Interface.

Please replace the paragraph on page 6, beginning on line 6, with the following paragraph.

In object-oriented programming, an object is a software bundle of related variables and methods. The present invention functions upon the interaction of three components, an object map, a class loader and a garbage collector. The object map quickly and inexpensively maps objects from the native side to the managed side. In the preferred embodiment, discussed infra, the map is a Standard Template Library map implemented as N-ary trees. The object map of the present invention has a key, which is the native object ID (in C++ parlance, this is the object pointer) and a value, which is the managed side object ID (in [[Java]]JAVA Native Interface parlance, this is the jobject).

Please replace the paragraph on page 6, beginning on line 14, with the following paragraph.

The class loader component is responsible to identify and load for execution the native class factory responsible to instantiate objects of that particular class. In the preferred embodiment discussed below, this loading is done on demand, that is, whenever an object of a particular class "crosses" from the managed side into the native side for the first time. In other words, the first time the class loader is asked to instantiate a native side object and the corresponding class factory is not loaded, it will load the executable module, perform any class initialization, and only then ask that class factory to instantiate the object. The class loader may decide to unload class factories based on criteria inherent to the implementation of the class loader. This will be appreciated by those skilled in the art as standard techniques to the [[Java]]<u>JAVA</u> programming language. In the preferred embodiment, class factories are never unloaded and will only cease to exist upon system termination, i.e., shutdown. Alternative methods, readily apparent to one skilled in the art, may be used for loading and unloading the class factories. Such methods include, but need not be limited to, loading in the background thread, loading related class factories, unloading after a period of inactivity, unloading the Least Recently Used factories, and unloading the Least Frequently Used factories.

Please replace the paragraph beginning on page 6, line 29, and ending on page 7, line 2, with the following paragraph.

The garbage collector disposes the references to native side objects when their managed side counterparts go out of scope. That is, the garbage collector removes pointers to native side objects when the managed side counterparts are removed during the normal garbage collection process inherent to [[Java]]<u>JAVA</u> programming. In the preferred embodiment, the garbage collector will activate to clean memory every five (5) seconds.

Please replace the paragraph on page 7, beginning on line 16, with the following paragraph.

Turning to Fig. 3, there is shown a map C for translating from a [[Java]]JAVA object 30 to a C++ object 34. While Fig. 3 denotes [[Java]]JAVA and C++, it will be appreciated by one skilled in the art that the map C may be applied to other object oriented programming languages. The [[Java]]JAVA, or managed side, object 30 is given in the form of an instance of jobject type. Integral to the [[Java]]JAVA object 30 is a pointer 32, which comprises a variable having a predefined value. When the value of this variable is non-zero, it is interpreted as a valid C++ object pointer 32. This allows the [[Java]]JAVA object 30, through the [[Java]]JAVA Native Interface (JNI) to invoke the method contained in the C++ object 34.

Please replace the paragraph beginning on page 7, line 24, and ending on page 8, line 2, with the following paragraph.

Fig. 4 shows the map for translating from C++ to [[Java]]JAVA, or rather, the map required for a C++ program to invoke a method or retrieve a variable existing in a [[Java]]JAVA program. The map 44 contains a key 46 and a corresponding value 48. It will be appreciated by one of ordinary skill in the art that while only a single key 46 and a single value 48 are shown, the method contemplated herein is applicable to a plurality of keys and corresponding values. The use of a single key 46 and a single value 48 are for exemplification purposes only. Returning to Fig. 4, there are a C++ object 40 and a [[Java]]JAVA object 42. To facilitate the translation from C++ to [[Java]]JAVA, the map 44 utilizes the native object ID (C++ object pointer as key 46) and the managed side object ID (JNI jobject or value 48). As each key 46 only points to a single C++ object 40 and each value 48 denotes a single [[Java]]JAVA object 42, the map is able to translate between the two programming languages.

Please replace the paragraph on page 8, beginning on line 3, with the following paragraph.

When the [[Java]]JAVA object 42 invokes a method or variable existing in a legacy program, such as the method or variables contained in the C++ object 40, a value 48 is registered

by the map 44. The map 44 queries the stored keys to determine which key, if any, corresponds to the value 48 of the [[Java]]JAVA object 42. Once the map 44 determines that key 46 corresponds to value 48, the map activates key 46. The key 46 points to the C++ object 40, whereby the method or variable sought by the [[Java]]JAVA object 42 is retrieved.

Please replace the paragraph on page 8, beginning on line 9, with the following paragraph.

The foregoing simplistic description is intended to provide an overview of the method and apparatus as is now going to be exemplified with references to the remaining figures. Turning to Fig. 5, there is shown a flow chart depicting the operation of the present invention as it translates, or maps, [[Java]]JAVA objects to C++ objects and invokes the methods contained therein. It will be appreciated that the use of [[Java]]JAVA and C++ are for purposes of explanation only and should not be construed as to limit the method solely to either [[Java]]JAVA or C++ translations. One of skill in the art will appreciate that the method described hereinafter, while denoting [[Java]]JAVA and C++, may be applied to any object-to-object oriented languages.

Please replace the paragraph on page 8, beginning on line 17, with the following paragraph.

A [[Java]]JAVA, or managed, side object is given in the form of an instance of jobject type. This instance is used to resolve a predefined [[Java]]JAVA object member variable, nativeCtx. Beginning at 502, using JNI calls, the value of nativeCtx is retrieved. The means for retrieval using JNI calls are well known in the art and one of appreciable skill will understand its use herein. The method progresses to a determination step at 504, in that the value of nativeCtx must be determined. For example, a zero value denotes that the class in which nativeCtx belongs has not before been encountered. A non-zero value denotes that the value retrieved in 502 represents a valid C++ object pointer. A determination in step 504 that a non-zero value was retrieved in 502 progresses the method to step 506, where the value is interpreted as a valid C++ object pointer. At step 508, the valid C++ object pointer is used to call the corresponding

method, i.e., the C++ object to which it refers. At step 510, the C++ object type and method name are resolved at compile time through the use of the appropriate JNI wrapper routine. The method proceeds to step 522, where the value of nativeCtx is updated to reflect an active reference, inhibiting the removal of the value during garbage collection.

Please replace the paragraph on page 9, beginning on line 1, with the following paragraph.

Returning to step 504, when the value of nativeCtx is determined to be zero, the [[Java]]JAVA object has not yet passed to the native (C++) side. In the case, the jobject instance is used to determine the [[Java]]JAVA class name of the object at 512. The class name, so determined, is then passed to the class loader at 514, which is asked to return the corresponding class factory. The class loader, as shown at 516, may load and initialize the class factory before returning it. One of ordinary skill in the art will recognize that the initializing of the class factory may occur subsequent to the return of the class factory. The loaded and initialized class factory is then returned to the program at step 518. At 520, the native side object is instantiated into a native side object, i.e., the original [[Java]]JAVA jobject has been translated into a corresponding C++ object. The value of nativeCtx is then updated at step 522. It will be appreciated by those skilled in the art that the updating of the value of nativeCtx equates to using JNI methods to contain the C++ object pointer in the [[Java]]JAVA object. Thus, the program has mapped [[Java]]JAVA to a C++ object and invoked the JNI method of the present invention.

Please replace the paragraph on page 9, beginning on line 14, with the following paragraph.

The mapping of a C++ object to [[Java]]JAVA and returning the object is exemplified in the flow chart of Fig. 6. A C++ object reference is returned to the program at step 602 in the form of a "smart pointer". The smart pointer is an object reference management tool, which replaces the normal C++ object pointer with an object reference. The lightweight object reference that is the smart pointer is then queried at step 604 for the ID of the object ID to which it refers. That is, the smart pointer is asked to divulge the C++ object pointer to which it refers.

Having thus retrieved the key at 604, the program progresses to determine that the key exists in the map at 606. In the event that the key does not exist in the map, the program will return to step 512 and progress from there. Returning to a positive determination in step 606, the value corresponding to the key is retrieved at 608, thereby giving the program the jobject instance. The jobject ID is then returned at 610. Thus, the C++ object has been mapped to [[Java]]<u>JAVA</u> and returned.

Please replace the paragraph on page 9, beginning on line 25, with the following paragraph.

The foregoing direct implementation of native C++ code, depicted in Figs. 5 and 6, allows for the C++ code to have a structure very closely resembling the inherent [[Java]]<u>JAVA</u> structure. This type of implementation is suitable for Application Programmer Interface (API) bindings, where the implementation is entirely accomplished on the C++ native side and the [[Java]]<u>JAVA</u> side is merely a binding (wrapper) to the API.

Please replace the paragraph beginning on page 9, line 30, and ending on page 10, line 11, with the following paragraph.

Returning first to Fig. 5, at step 514, the class name has been sent to the class loader for retrieval of the corresponding class factory. Thus the class loader has been asked to return a class factory for a particular managed [[(Java)]]<u>(JAVA)</u> class name (the class name being based upon the jobject type). Turning now to Fig. 7, there is shown a flow chart for the method by which the class loader retrieves and returns the information requested. The class loader receives the request to return a class factory for a particular managed [[(Java)]]<u>(JAVA)</u> class name at step 702. The class loader then maps the managed [[(Java)]]<u>(JAVA)</u> class name into a native (C++) class name at step 704. It will be understood by those skilled in the art that the class loader of the present invention may be utilized in any object-to-object program, as object-oriented programming utilizes the class layout/function. The class loader must then determine, at 706, if the class factory corresponding to the native class name mapped in 704, has already been loaded. In the

event that the corresponding class factory has previously been loaded, the class loader then returns the class factory to the program at 718.

Please replace the paragraph on page 10, beginning on line 20, with the following paragraph.

The C++ class factories can thereby be loaded on demand whenever instances of particular classes are required or whenever the [[Java]]<u>JAVA</u> class loader loads the particular package. This is particularly useful for the more modular structure of C++ code, which is dynamically loaded only when required. Thus, the program has retrieved and returned a class factory by using a class name.

Please replace the paragraph beginning on page 10, line 25, and ending on page 11, line 11, with the following paragraph.

Inherent to the [[Java]]<u>JAVA</u> programming language, a garbage collection occurs to remove unused objects and improve memory utilization. However, the present invention furthermore provides a native-side garbage collector, or simply garbage collector, which collects the C++ references. That is, when a [[Java]]<u>JAVA</u> object is removed from under-utilization, the corresponding C++ smart pointer is also collected and removed. Turning now to Fig. 8, there is shown a flow chart for the operation of the garbage collector of the present invention. The process for collecting the unused objects and references begins at 802 when the garbage collector awakens. It will be appreciated by those skilled in the art that [[Java]]<u>JAVA</u> allows for the manual activation, as well as the automatic activation, of the garbage collector. In this, the preferred embodiment, the garbage collector is set to activate every five (5) seconds to dispose of unused items. The garbage collector starts in step 804 to iterate the object map for each [[Java]]<u>JAVA</u> jobject. Upon completion in step 804, the garbage collector must then determine at step 806, if each reference is or is not weak. Weak is a [[Java]]<u>JAVA</u> term-of-art for an out-of-scope or not used object or reference. A non-weak reference will be left alone. In the event that the garbage collector finds a weak reference, it will collect the native-side reference at step 808. For purposes herein, the reference so collected is a C++ smart pointer. The smart pointer, or

reference, is then disposed of resulting in step 810 of a decrease in the reference count of the C++ object. This decrease may eventually lead to the removal of the C++ object itself, if the reference count drops to zero.

Please replace the paragraph on page 11, beginning on line 12, with the following paragraph.

The end result of the garbage collection is the removal and disposal of all C++ objects that have had their [[Java]]JAVA side counterparts go out-of-scope. The [[Java]]JAVA garbage collector may still not collect the reference at this time, however, the native resources (C++ objects) associated with it are released. The "nativeCtx" example is never updated, since [[Java]]JAVA references are never re-promoted to active. After completing the collection and disposal process, the garbage collector will proceed to step 812 and "sleep" until it is activated again in five (5) seconds.

Please replace the paragraph on page 11, beginning on line 18, with the following paragraph.

Consequently, memory allocated by the native side is managed separately from the memory allocated from the [[Java]]JAVA (managed) side. Since [[Java]]JAVA garbage collection is notoriously conservative, the net result is a sizeable improvement in memory utilization.

Please replace the Abstract of The Disclosure with the following paragraph:

This invention is directed to a method and apparatus for implementing native code directly in C++ with object structure very closely resembling the object structure of [[Java]]JAVA. The method provides for the mapping of [[Java]]JAVA to C++ objects and invoking the [[Java]]JAVA Native Interface, mapping C++ objects to [[Java]]JAVA and returning a corresponding [[Java]]JAVA object, retrieving and returning a class factory by class name, and collecting garbage in both the C++ and [[Java]]JAVA environments.